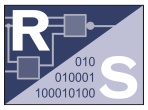
Implementierung und Optimierung eines On-Chip Routing Verfahrens für SpartanMC

Bachelorarbeit

Felix Karl Sterzelmaier, Bensheim

15. April 2016

TECHNISCHE UNIVERSITÄT DARMSTADT

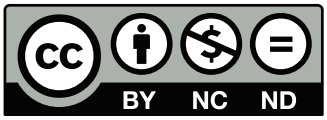


Erklärung gemäß § 22 Abs. 7 APB

Hiermit erkläre ich gemäß § 22 Abs. 7 der Allgemeinen Prüfungsbestimmungen (APB) der Technischen Universität Darmstadt in der Fassung der 4. Novelle vom 18. Juli 2012, dass ich die Arbeit selbstständig verfasst und alle genutzten Quellen angegeben habe und bestätige die Übereinstimmung von schriftlicher und elektronischer Fassung.

Darmstadt, den 15. April 2016	
Ort, Datum	Name

Fachbereich Elektro- und Informationstechnik
Institut für Datentechnik
Fachgebiet Rechnersysteme
Prüfer: Prof. Dr.-Ing. Christian Hochberger
Prüfer: Prof. Dr.-Ing. Andreas Koch
Betreuer: M.Sc. Kris Heid



Inhaltsverzeichnis

1 Einleitung

Dieses Thema wurde 2015 bereits in der Bachelorarbeit [flaig] „Design and Implementation of an On-Chip Routing Method for SpartanMC[1]“ von Maximilian Flaig behandelt. Jedem Leser wird nahegelegt, seine Arbeit zuerst zu lesen, da Grundkenntnisse über das Thema beschrieben werden, die hier nur verkürzt dargestellt werden.

1.1 Motivation

Die bisherigen Kommunikationsmöglichkeiten mehrerer SpartanMC Instanzen auf einem FPGA[2] sind zwar performant, benötigen jedoch viele Ressourcen (Register und LUT[3]) aufgrund der direkten Verbindung aller miteinander kommunizierenden Module. Vor allem dann, wenn nur wenige Nachrichten ausgetauscht werden sollen, bei denen es weniger auf Geschwindigkeit ankommt ist das Kosten-Nutzen-Verhältnis nicht optimal. Es soll eine Möglichkeit gefunden werden, auf Kosten der Geschwindigkeit weniger Platz zu verbrauchen.

1.2 Aufgabenbeschreibung

Für den SpartanMC soll ein Routermodul geschrieben werden, das mehrere Kerne verbindet und weniger Ressourcen als ein Aufbau mit den bisherigen Core Connectoren benötigt (die nur zwei SpartanMC-Kerne direkt verbinden können). Die Daten sollen an wenigen Orten zwischengespeichert werden, um eine möglichst performante Kommunikation sicherzustellen. Beim Aufbauen der Verbindung muss jeder Router erkennen, wohin die Daten adressiert sind. Außerdem soll der Empfänger erfahren, von wo die Daten verschickt wurden. Jeder Router mit mehreren Eingängen soll das Round-Robin-Verfahren verwenden, um alle Eingänge mit der gleichen Priorität zu bearbeiten.
Des Weiteren sollen Tests mit mindestens fünf Routern durchgeführt und eine Demonstrationsfirmware bereitgestellt werden, mit der die Verwendung des Moduls ersichtlich ist. Evaluiert werden sollen die Vor- und Nachteile im Vergleich zum Core Connector Modul. Hierbei soll ein Schwerpunkt bei Durchsatz, Latenz und Ressourcennutzung gesetzt werden.

[1] Spartan Microcontroller
[2] Field Programmable Gate Array
[3] Lookup table

2 Analyse des bereits bestehenden Routermoduls

Von Maximilian Flaig lagen sowohl die Bachelorarbeit, als auch der Quellcode vor. Eine Dokumantation im SpartanMC-Manual war nicht vorhanden. Bei der Untersuchung des Routermoduls stellte sich heraus, dass dieses nicht funktionsfähig ist. Kommentare im Code waren kaum vorhanden. Der Schaltplan wirkt unübersichtlich, da nur selten Funktionalität in Submodule ausgegliedert wurde. Das Prinzip Teile-und-Herrsche wurde kaum angewandt. Das Verstehen und Bewerten der vorliegenden Materialien fiel hierduch schwer.
Positiv aufgefallen ist die Funktionalität des C-Codes. Hiervon wurden Teile übernommen. Auch das Format des Headerpakets wurde beibehalten.

3 Neues Routermodul

Es wurde beschlossen, dass sich das Neuschreiben effektiver gestaltet, als das Verstehen und Optimieren des bestehenden Moduls. Die Ziele waren neben der Funktionalität des gesamten Moduls ein übersichtlicheres Design und eine gute Dokumentation, was eine Verwendung und Weiterentwicklung vereinfachen soll. Es wurde teilweise auch Code des alten Routermoduls wiederverwendet.

3.1 Designentscheidungen

Es wurde ein Blocking-Router verwendet, um die Komplexität gering zu halten. Dieser kann nur eine Nachricht gleichzeitig verarbeiten. Das bedeutet: Wenn mehr als eine Anfrage anliegt, werden nicht alle angenommen, sondern manche abgewiesen. Die Anzahl der Ein- und Ausgänge ist variabel und in JConfig (Einer für SpartanMC entwickelten Software, mit der einzelne Module auf einem FPGA platziert und verbunden worden können) einstellbar, ebenso die Buffergrößen. Hierdurch werden dem Anwender Freiheiten bei der Verbindung mehrerer SpartanMC-Kerne geboten. Die Routingtabellen sind statisch und werden ebenfalls in JConfig eingetragen. Am Headerpaket wurde nichts geändert. Es besteht weiterhin aus je sechs Bits für Sender, Empfänger und Größe der Nachricht. Eine Nachricht kann nur gesendet werden, wenn im Buffer von Sender und Empfänger genügend Platz vorhanden ist, um die gesamte Nachricht und das Headerpaket zu speichern. Ist die Verbindung aufgebaut, wird die Nachricht am Stück übertragen. Um Deadlocks zu vermeiden, wurde ein Zähler im Sender eingebaut. Wird eine Nachricht abgewiesen, wartet der Sender eine gewisse Zeit, bevor er einen weiteren Versuch unternimmt. Die Dauer ist bei jedem Router verschieden, da sie aus der Routernummer abgeleitet wird. Das Layout soll möglichst einfach gehalten werden und leicht verständlich sein, damit keine großen Hürden für die Weiterentwicklung aufgestellt werden. So wird das Prinzip Teile-und-Herrsche angewendet und Funktionalität in Submodule ausgegliedert. Wenn ein Router keine Ausgänge besitzt, wird das Modul Sender nicht synthetisiert, um Platz auf dem FPGA zu schaffen. Ebenso wird das Modul Receiver nicht synthetisiert, wenn keine Eingänge existieren.

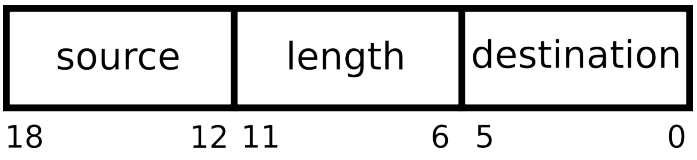


Abbildung 3.1: Headerpaket

3.1.1 Leitungen

Wenn 2 Router miteinander verbunden werden, erfolgt das immer mit 3 Leitungen:

- data: 18 Bits, von Sender zu Empfänger. Hierüber werden die Daten übertragen. Während des Verbindungsaufbaus liegt hier das Headerpaket an.
- request: 1 Bit, von Sender zu Empfänger. Hierüber wird angezeigt, dass ein Router Daten senden möchte. Die Leitung bleibt „1“, bis die Übertragung abgeschlossen ist oder ein Router die Verbindung ablehnt.
- return: 2 Bits, von Empfänger zu Sender. Hierüber wird angezeigt, ob ein Verbindungsaufbau akzeptiert oder abgelehnt wird.

Bitmuster	Bedeutung
00	noch nicht verarbeitet
01	abgelehnt
10	akzeptiert
11	Übertragung beendet

Abbildung 3.2: Bitmuster der Returnleitung

3.1.2 Layout

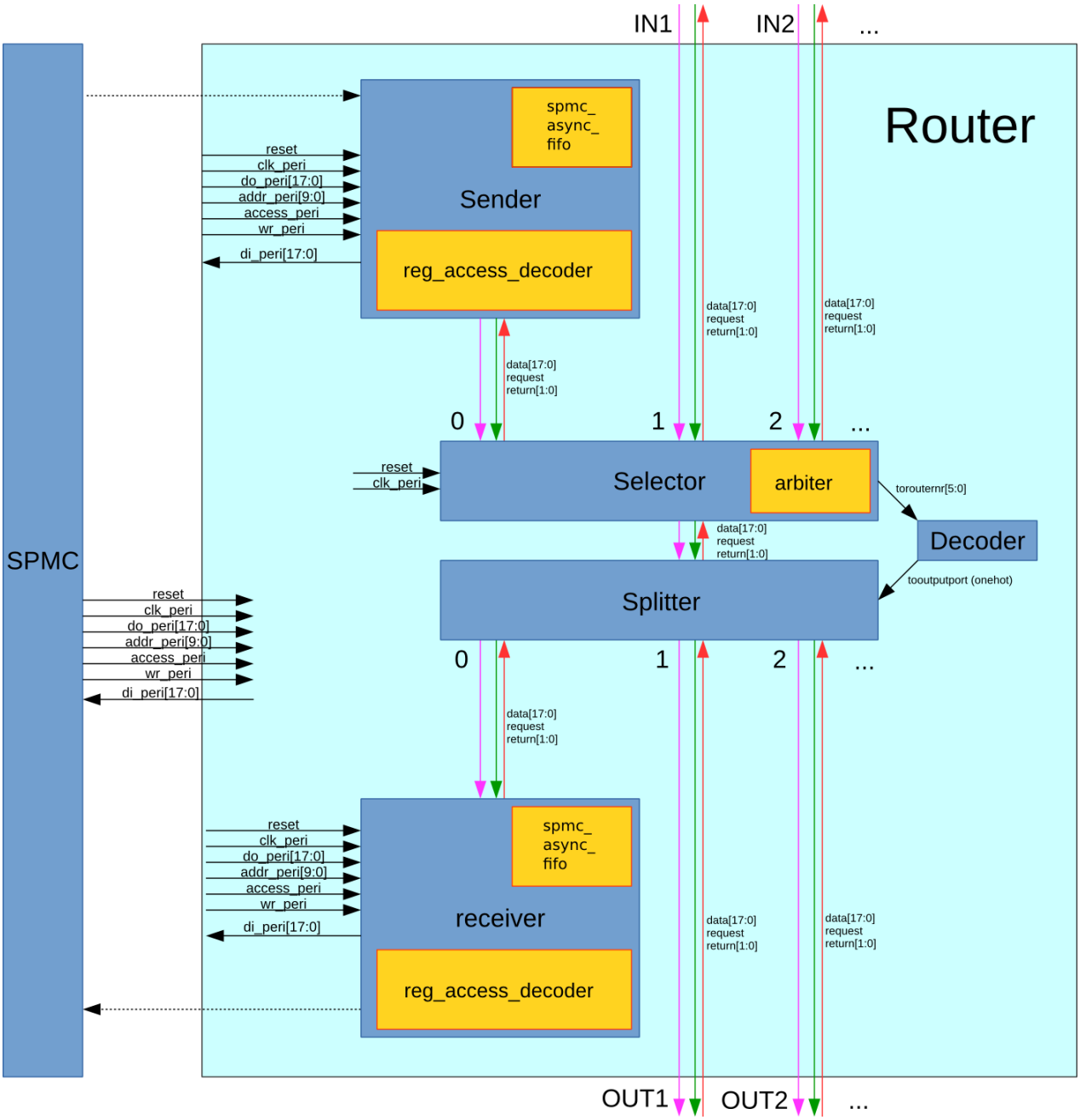


Abbildung 3.3: Routerlayout

- Das Topmodul Router enthält weitestgehend keine Logik, sondern verbindet die Submodule, Ein- und Ausgänge miteinander.
- Sender: Dieses Modul besitzt Verbindungen zum SpartanMC-Kern sowie zum Selector. Es wird nur synthetisiert, wenn der Router Ausgänge besitzt. Es benutzt die Untermodule „reg_access_decoder“ für die Kommunikation mit dem Kern sowie „spmc_async_fifo_wrapper“ für den Buffer. Wenn der Kern Daten versenden möchte, schreibt er diese in den Buffer. Um alles Weitere kümmert sich der Sender. Er versucht, eine Verbindung aufzubauen und wenn dies gelingt, versendet er die Daten.
- Selector: Hier laufen alle Eingänge und der Sender zusammen. Mit dem Untermodul Arbiter wird das Round-Robin-Verfahren verwendet, um gleichberechtigt einen der Eingänge oder den Sender auszuwählen, sofern die Requestleitung „1“ ist. Der Eingang oder Sender wird dann an die Ausgänge Richtung Splitter weitergeleitet. Parallel wird die Routernummer des Empfängers zwischengespeichert und an den Decoder weitergegeben. Bei allen nicht gewählten Eingängen wird return auf „01“ gesetzt. Wenn beim gewählten Eingang request auf „0“ fällt, wird in den Startzustand gewechselt.
- Decoder: Enthält die in JConfig eingetragene Routingtabelle. Weist eine Binärcodierte Routernummer einem One-Hot-codierten Ausgang zu.
- Splitter: Erhält vom Selector die drei Leitungen „data“, „request“ und „return“ und leitet diese an den passenden Ausgang, oder den Receiver, weiter. Die Ausgangsnummer wird One-Hot-codiert vom Decoder übermittelt.
- Receiver: Dieses Modul besitzt Verbindungen zum SpartanMC-Kern sowie zum Splitter. Es wird nur synthetisiert, wenn der Router Eingänge besitzt. Es benutzt die Untermodule „reg_access_decoder“ für die Kommunikation mit dem Kern sowie „spmc_async_fifo_wrapper“ für den Buffer. Es empfängt Daten und reicht diese anschließend an den Kern weiter.
- spmc_async_fifo_wrapper: Erweitert das Untermodul „spmc_async_fifo“ um einen Zähler. Dieser enthält die Information, wie viele Buffereinträge belegt sind.

3.1.3 Routingtabelle

Die Routingtabelle, welche für jeden Router in JConfig individuell eingetragen werden kann, dient der Zuordnung von Empfänger zu einem Ausgang. Sowohl Routernummern als auch Ausgänge sind ganze Zahlen, beginnend bei 0. Für alle nicht erreichbaren Router wird 0 eingetragen. In der Praxis würden solche Pakete dann zu falschen Routern geleitet werden. Der Benutzer ist dafür verantwortlich, dass dieser Fall nie eintritt.

Ausgang	Bedeutung mit Receiver	Bedeutung ohne Receiver
0	eigener Receiver	1. Ausgang
1	1. Ausgang	2. Ausgang
2	2. Ausgang	3. Ausgang
3	3. Ausgang	4. Ausgang

Abbildung 3.4: Beschreibung der Routingtabelle

3.2 Implementierung

3.2.1 Router

Das Routermodul ist das Hauptmodul, das alle folgenden Submodule verwendet. Es besitzt Verbindungen zum SpartanMC-Core sowie die Leitungen „data“, „request“ und „return“ von und zu anderen Routern. Hier werden in zwei Generate-Blöcken die Leitungen von Sender und Eingängen zu Selector sowie die von Splitter zu Receiver und Ausgängen zusammengefasst. Als Eingang ist hier das Leitungsbündel „datain“, „requestin“ und „returnout“ gemeint. Als Ausgang „dataout“, „requestout“ und „returnin“. Der Begriff Eingang und Ausgang ist für die Returnleitung eigentlich nicht zutreffend, da sie in die entgegengesetzte Richtung verläuft.

3.2.2 Sender

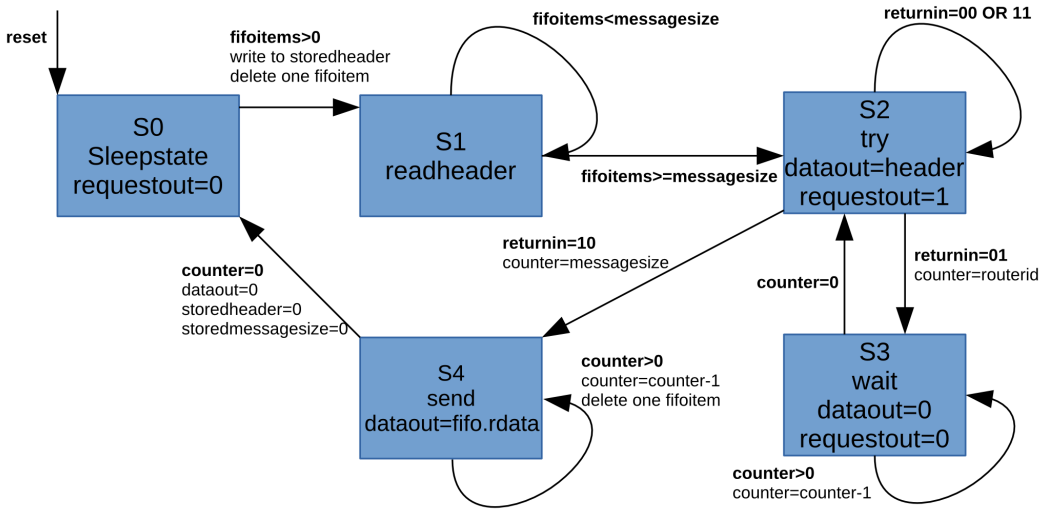


Abbildung 3.5: Endlicher Automat des Senders

Neben den Verbindungen zum SpartanMC-Kern ist dieses Modul mit dem Selector über die Leitungen „dataout“, „requestout“ und „returnin“ verbunden. Es ist als endlicher Automat mit den fünf Zuständen S0-S4 implementiert. Die Verbindung „di_peri“ zum Kern wird zusammen mit dem Receivermodul verwendet. Sie darf beim Kern nie hochohmig sein, da dieser sonst Berechnungen nicht korrekt durchführt. Deshalb verfügt der Sender über die Information, ob in diesem Router ein Receivermodul existiert. Wenn der Sender nicht vom Kern angesprochen ist und ein Receivermodul existiert, setzt der Sender „di_peri“ hochohmig. Existiert kein Receivermodul, wird es auf „0“ gesetzt.
Nachdem sich der Kern vergewissert hat, dass genügend freie Buffereinträge vorhanden sind, schreibt er das Headerpaket sowie die Nachricht in den Buffer. Der Automat befindet sich zu Beginn in S0. Sobald der Buffer mindestens einen Wert enhält (das Headerpaket), wird in S1 gewechselt und der Teil des Werts, der die Nachrichtenlänge enthält, im Register „messagesize“ gespeichert. Nun wird gewartet, bis der Buffer mindestens so viele Einträge enthält, wie die Nachricht lang ist. Ist dies der Fall, wird in S2 gewechselt. Nun wird versucht, eine Verbindung

aufzubauen, indem das Headerpaket auf die Datenleitungen und „requestout“ auf „1“ gesetzt werden. Wenn auf der Returnleitung „01“ anliegt, wird in S3 gewechselt, bei „10“ in S4.
In S3 wird gewartet, um Deadlocks zu vermeiden. Anschließend wird wieder in den Zustand S2 gewechselt, um eine Verbindung aufzubauen.
In S4 wird in jedem Takt ein Teil der Nachricht übertragen. Die Requestleitung bleibt hierbei auf „1“. Wenn die Übertragung abgeschlossen ist, wird in den Ausgangszustand S0 gewechselt und die Daten- sowie Requestleitungen werden auf „0“ gesetzt.

3.2.3 Selector

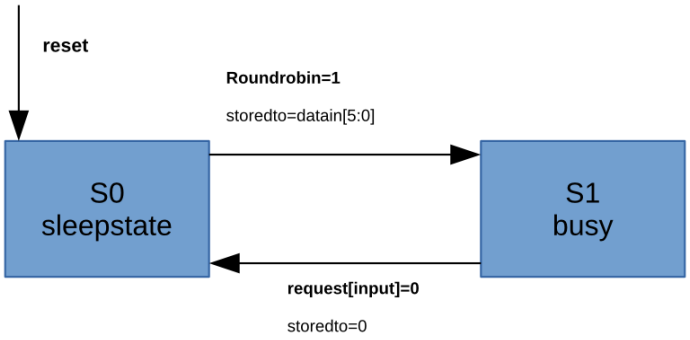


Abbildung 3.6: Endlicher Automat des Selectors

Dieses Modul entscheidet, welche Daten verarbeitet werden. Es ist mit den Eingängen, dem Sender, Decoder und Splitter verbunden und ebenfalls als endlicher Automat implementiert. Dieser besitzt die zwei Zustände S0 und S1. Alle Requestleitungen der Eingänge führen in das Untermodul Arbiter, welches das Round-Robin Verfahren implementiert. Sobald der Arbiter einen Eingang gewählt hat, wird in S1 gewechselt, die Nummer des empfangenden Routers aus dem Headerpaket extrahiert und im Register „storedto“ gespeichert. In S1 werden die drei Leitungen „data“, „request“ und „return“ des gewählten Eingangs durchgereicht und „return“ an allen anderen Eingängen auf „01“ gesetzt, was „ablehnen“ bedeutet. Die gespeicherte Routernummer liegt am Ausgang „routernr“ an, welcher mit dem Decoder verbunden ist. Sobald am gewählten Eingang „request“ auf 0 fällt, wird der Arbiter zurückgesetzt, das Register „routernr“ auf „0“ gesetzt und in den Zustand S0 gewechselt. Der Selector ist also mit einem Multiplexer vergleichbar, der seinen Eingang selbst wählt.

3.2.4 Decoder

Der Decoder enthält ausschließlich kombinatorische Logik und wandelt die Routernummer des Empfängers in eine One-Hot-Codierte Nummer des zu verwendenden Ausgangs am derzeitigen Router um. Diese wird an das Splittermodul weitergegeben. Er enthält die in JConfig eingetragene Routingtabelle.

3.2.5 Splitter

Äquivalent zum Selector ist der Splitter mit einem Demultiplexer vergleichbar. Er verbindet die drei vom Selector kommenden Leitungen „data“, „request“ und „return“ mit dem gewählten Ausgang oder dem Receiver. Welche dieser Möglichkeiten gewählt wird, ist vom Eingang „tooutputport“ abhängig, welcher mit dem Decodermodul verbunden ist.

3.2.6 Receiver

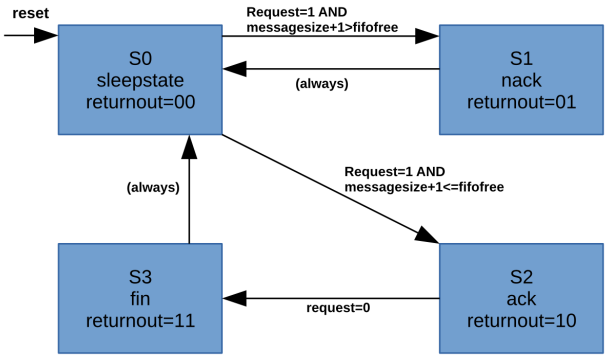


Abbildung 3.7: Endlicher Automat des Receivers

Neben den Verbindungen zum SpartanMC-Kern ist dieses Modul mit dem Splitter über die Leitungen „datain“, „requestin“ und „returnout“ verbunden. Es ist als endlicher Automat mit den vier Zuständen S0-S3 implementiert. Die Verbindung „di_peri“ zum Kern wird zusammen mit dem Sendermodul verwendet. Sie darf beim Kern nie hochohmig sein, da dieser sonst Berechnungen nicht korrekt ausführt. Deshalb verfügt der Receiver über die Information, ob in diesem Router ein Sendermodul existiert. Existiert ein Sender und ist dieser angesprochen, so wird „di_peri“ hochohmig gesetzt. Existiert kein Sendermodul, wird es auf 0 gesetzt.

Zu Beginn befindet sich der Receiver im Zustand S0 und „returnout“ ist auf „00“ gesetzt. Sobald „request“ auf „1“ wechselt, gibt es zwei Möglichkeiten: Entweder die Nachricht inklusive Headerpaket passen noch in den Buffer, dann wird in den Zustand S2 gewechselt und „returnout“ auf „10“ gesetzt. Oder das ist nicht der Fall, dann wird in S1 gewechselt und „returnout“ auf „01“ gesetzt. Dieser Zustand wird immer nach einem Takt wieder verlassen. In S2 wird in jedem Takt ein Wert der Leitung „datain“ in den Buffer geschrieben, bis „request“ auf „0“ fällt. Dann wird in S3 gewechselt, in dem „returnout“ auf „11“ gesetzt wird. Nach einem Takt wird dann automatisch in den Startzustand S0 übergegangen.

3.2.7 spmc_async_fifo_wrapper

Das bereits bestehende Modul „spmc_async_fifo“ wurde um einen Zähler erweitert. Dieser wurde größtenteils aus der Datei „core_connector_master.v“ aus dem Gitbranch „dma“ übernommen. Es stellte sich jedoch heraus, dass der Zähler einen Takt schneller war als der Fifo, an dessen Ausgang erst später der eigentliche Wert anlag. Der Sender wechselte daher zu früh

in den Zustand S1 und las einen falschen Wert als Headerpaket. Um dieses Problem zu umgehen, wurde der Zähler um einen Takt verzögert. Beim Testen traten daraufhin keine Fehler mehr auf. Es könnte jedoch unter Umständen möglich sein, dass bei zwei aufeinanderfolgenden Nachrichten, die vom Kern in den Buffer geschrieben werden, bei der zweiten ein Wert verloren geht.

3.2.8 define.v

Dies ist kein Modul, sondern enthält lediglich die Hilfsfunktion LOG2[1]. Diese wird während der Synthetisierung verwendet und berechnet den Logarithmus zu Basis 2. Mit ihr kann aus der Buffergröße die Anzahl der benötigten Adressierungsbits berechnet werden.

3.2.9 Register

Die Kommunikation zwischen Kern und Router erfolgt über die drei Register „data“, „free_entries“ und „data_available“. Diese werden wie folgt definiert:

```
typedef struct {
    volatile unsigned int data;
    volatile unsigned int free_entries;
    volatile unsigned int data_available;
} router_regs_t;
```

Abbildung 3.8: Definition der Routerregister

Über das Dataregister werden Daten vom Kern zum Sender und vom Receiver zum Kern übertragen. Free_entries benutzt der Kern, um zu erfahren, wieviele freie Buffereinträge im Sender vorhanden sind. Nur wenn die gesamte Nachricht inklusive Headerpaket im Buffer Platz findet, wird der Kern die Daten kopieren. Data_available verwendet der Kern, um vom Receiver zu erfahren, wieviele Nachrichtenpakete bereits empfangen wurden, die noch nicht zum Kern kopiert wurden.

[1] Zweierlogarithmus

3.2.10 Sender C-Code

Zum Senden von Daten kann folgende Funktion verwendet werden:

```
void router_send_data(router_regs_t *router, void *data, unsigned int
    source, unsigned int msgsize, unsigned int dest);
```

Parameter	Bedeutung
*router	Pointer auf die Routerregister
*data	Pointer auf ein Array der zu sendenden Daten
source	Die Nummer des sendenden Routers
msgsize	Die Nachrichtenlänge
dest	Die Nummer des Empfangsrouters

Abbildung 3.9: Funktion zum Senden

3.2.11 Receiver C-Code

Mit dieser Funktion kann überprüft werden, wieviele Daten der Receiver empfangen hat, die noch nicht zum Kern kopiert wurden. Der Parameter *router ist ein Pointer auf die Routerregister.

```
unsigned int router_check_data_available(router_regs_t *router);
```

Abbildung 3.10: Funktion zum Überprüfen auf empfangene Daten

Die Funktion „router_read“ kann zum Einlesen einer Nachricht verwendet werden. Sie erwartet als erstes Paket den Nachrichtenheader und liest danach die Anzahl an Paketen ein, die in diesem angegeben sind.

```
void router_read(router_regs_t *router, void *data, unsigned int *
    msgsize, unsigned int *source);
```

Parameter	Bedeutung
*router	Pointer auf die Routerregister
*data	Pointer auf ein Array. Dieses muss groß genug sein, um alle Daten aufnehmen zu können.
*source	Pointer auf die Nummer des sendenden Routers
*msgsize	Pointer auf die Nachrichtenlänge

Abbildung 3.11: Funktion zum Empfangen

3.3 Integration und erstellte Werkzeuge

3.3.1 JConfig

Die JConfig Konfiguration für das Routermodul wurde angepasst. Als zusätzlicher Parameter wird die Routerid angezeigt, die für das Erstellen der Routingtabelle sowie die unterschiedlichen Wartezeiten zur Verbindungswiederholung benötigt wird. Bei den Verbindungen können nun „datain“, „requestin“, „returnin“, sowie „dataout“, „requestout“ und „returnout“ gesetzt werden. Die übrigen Parameter und Verbindungen wurden übernommen. Hierbei handelt es sich um die Leitungen zum SpartanMC Kern, die Kapazität der Buffer in Sender und Receiver, die Anzahl der Ein- und Ausgänge, sowie die Routingtabelle.

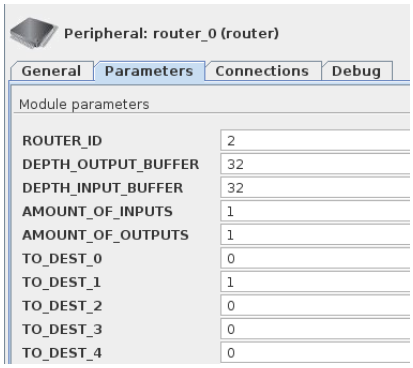


Abbildung 3.12: JConfig Parameter

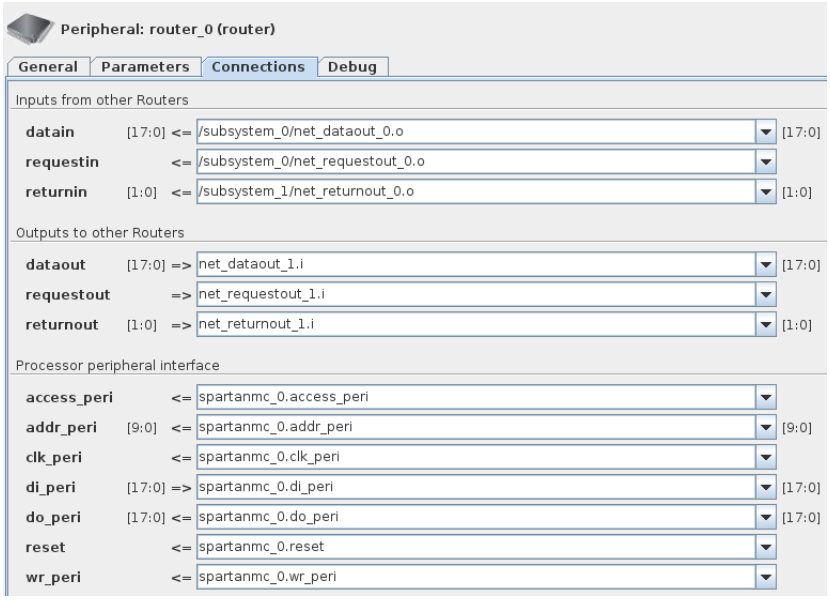


Abbildung 3.13: JConfig Verbindungen

3.3.2 Routingtable (Javatool)

Mit dem Befehl „make routing“ im Projektverzeichnis kann die Datei jconfig.xml eingelesen und verarbeitet werden. Diese muss jedoch bestimmten Anforderungen gerecht werden, welche dem SpartanMC-Benutzerhandbuch entnommen werden können. Folgende Ausgabedateien werden erzeugt:

- routingtable_debugoutput.txt: Hier werden unter anderem eventuell aufgetretene Fehler ausgegeben.
- routingtable_FromViaTo.txt: Die Verbindungen, die das Werkzeug festgestellt hat. Notation: von_routerid:splitter_ausgang:nach_routerid
- routingtable_graph.dot: Ein Graph für GraphViz. Mit folgendem Befehl kann eine svg[2]-Datei erzeugt werden:

  ```
  neato -Tsvg routingtable_graph.dot -o routingtable.svg -Gstart=rand
  ```

- routingtable_text.txt: Die Ausgabe des Dijkstra-Algorithmus. Diese Werte sollten in JConfig eingegeben werden.

3.4 Beispiel

Die Pfeile in Abbildung 3.15 beschreiben die Richtung der Data- und Requestleitung. Die Returnleitung verläuft in entgegengesetzter Richtung. In diesem Beispiel sind die meisten Verbindungen zwischen Routern in beide Richtungen ausgeführt. Lediglich die Router mit der Nummer zehn und elf besitzen nur einen Aus- bzw Eingang. Die Zahl am Ausgangspunkt eines Pfeiles beschreibt die Nummer des Ausgangs am Splitter. Da der Router zehn keinen Eingang und somit

[2] Scalable Vector Graphics

auch keinen Receiver enthält, ist hier „0“ der erste Ausgang. Bei allen anderen Routern ist „0“ am Receiver angeschlossen und „1“ am ersten Ausgang.

Subsystem 2	Subsystem 3	Subsystem 6	Subsystem 10	Subsystem 11
TO_DEST_0: 1	TO_DEST_0: 1	TO_DEST_0: 2	TO_DEST_0: 0	TO_DEST_0: 0
TO_DEST_1: 1	TO_DEST_1: 1	TO_DEST_1: 2	TO_DEST_1: 0	TO_DEST_1: 0
TO_DEST_2: 0	TO_DEST_2: 3	TO_DEST_2: 2	TO_DEST_2: 0	TO_DEST_2: 0
TO_DEST_3: 2	TO_DEST_3: 0	TO_DEST_3: 2	TO_DEST_3: 0	TO_DEST_3: 0
TO_DEST_4: 2	TO_DEST_4: 2	TO_DEST_4: 1	TO_DEST_4: 0	TO_DEST_4: 0
TO_DEST_5: 2	TO_DEST_5: 2	TO_DEST_5: 1	TO_DEST_5: 0	TO_DEST_5: 0
TO_DEST_6: 1	TO_DEST_6: 1	TO_DEST_6: 0	TO_DEST_6: 0	TO_DEST_6: 0
TO_DEST_7: 1	TO_DEST_7: 1	TO_DEST_7: 2	TO_DEST_7: 0	TO_DEST_7: 0
TO_DEST_8: 1	TO_DEST_8: 1	TO_DEST_8: 2	TO_DEST_8: 0	TO_DEST_8: 0
TO_DEST_9: 1	TO_DEST_9: 1	TO_DEST_9: 2	TO_DEST_9: 0	TO_DEST_9: 0
TO_DEST_10: 0	TO_DEST_10: 0	TO_DEST_10: 0	TO_DEST_10: 0	TO_DEST_10: 0
TO_DEST_11: 3	TO_DEST_11: 3	TO_DEST_11: 2	TO_DEST_11: 0	TO_DEST_11: 0

Abbildung 3.14: Beispiel Routingtable

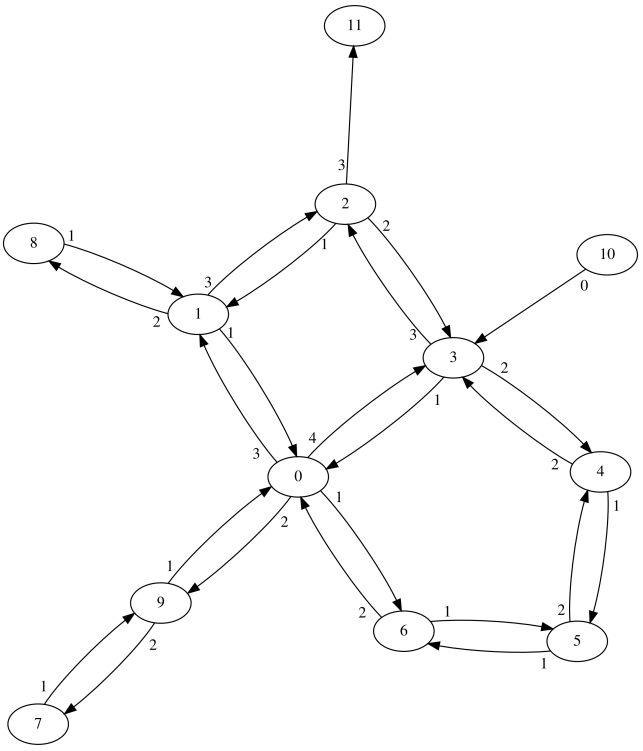


Abbildung 3.15: Beispielaufbau (SpartanMC-Kerne, mit Routermodulen verbunden)

4 Evaluation

In diesem Testaufbau fassten alle Buffer 32 Einträge. Als Zielplattform wurde das Nexys-Video mit dem FPGA XC7A200T-SBG484-1 gewählt. Die folgenden Kapitel 4.1, 4.2 und 4.3 beziehsen sich auf Netze mit zwei bis sechs SpartanMC-Kernen. Die Abbildungen 4.1 und 4.2 zeigen das Layout bei fünf Modulen. In beiden Szenarian können alle SpartanMC-Kerne miteinander kommunizieren. Beim Routerlayout wurde hierfür eine Ring-Topologie verwendet. Das Duplex Core Connector Layout ist hingegen auf ein voll vermaschtes Netz angewiesen.

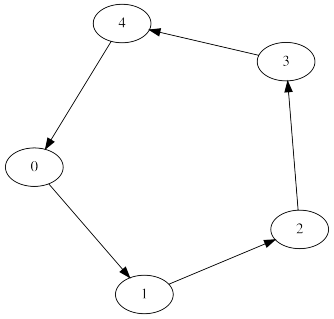


Abbildung 4.1: Routerlayout

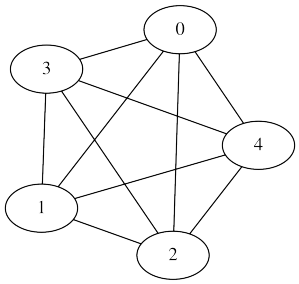


Abbildung 4.2: Duplex Core Connector Layout

4.1 Maximale Taktrate

Vorgehensweise: Es wurden Vielfache von 5MHz getestet. Der hier angegebene Wert ist der höchste, bei dem keine Fehler bei der Synthetisierung aufgetreten sind.

Anzahl	Router	Duplex Core Connector
2	60	55
6	55	45

Abbildung 4.3: Maximale Taktrate in MHz

Bei ledigleich zwei miteinander verbundenen Modulen besitzt das Routerlayout eine höhere Taktrate um 5MHz. Bei sechs Modulen kann dieser Vorsprung sogar auf 10MHz ausgebaut werden, was einer Verbesserung von 22 % entspricht.

4.2 Ressourcen

Anzahl	Router	Duplex Core Connector
2	752	618
3	1126	1111
4	1500	1728
5	1874	2469
6	2248	3334

Abbildung 4.4: Anzahl an Slice Register

Anzahl	Router	Duplex Core Connector
2	2176	2107
3	3251	3553
4	4376	5363
5	5397	7507
6	6532	10056

Abbildung 4.5: Anzahl an Slice LUT

Diese Werte wurden dem Synthetisierungsprotokoll entnommen. Bei zwei verbundenen Modulen benötigt der Duplex Core Connector weniger Platz auf dem FPGA. Ab vier Modulen ist der Router sparsamer. Dies liegt vor allem daran, dass der Core Connector ein voll vermaschtes Netz benötigt. Dies bedeutet, dass bei **n** SpartanMC-Kernen **n** Routermodule und **n*(n-1)** Core Connector Module benötigt werden.

4.3 Übertragungszeiten

Es wurden zehn Werte bei einer Taktfrequenz von 60MHz übertragen. Die Übertragungszeit zwischen den beiden SpartanMC-Kernen auf Hardwareebene wurde aus der Simulation entnommen. Beim Routermodul wurde die Zeit, in der die Requestleitung auf „1“ war gemessen, also die Zeit für Verbindungsaufbau und Übertragung. Beim Core-Connector wurde die Zeit gemessen, in der die zehn Werte an der Datenleitung anlagen.

Abstand	Router	Core Connector
1	466,6	633,3
2	533,3	
3	600,0	
4	666,6	
5	733,3	

Abbildung 4.6: Übertragungszeit in ns (Hardwareebene zwischen Modulen)

Beim Vergleich der beiden Module fällt auf, dass der Core-Connector mehr Zeit benötigt, obwohl er auf den Verbindungsaufbau verzichten kann. Dies hängt damit zusammen, dass diese Lösung für einen Wert zwei Takte, das Routermodul hingegen nur einen Takt benötigt. Diese Betrachtung blendet jedoch aus, dass die Lösung mit Routermodulen die Werte zweimal in einem Buffer zwischenspeichert und erst fortfährt, wenn alle Werte in diesem Buffer vorliegen. Deshalb wurde ein zweiter Test durchgeführt, in dem im C-Code des Senders eine Zeile vor dem Senden ein Wert ausgegeben wird, den man im Wellendiagramm der Simulation sehen kann. Ebenso wird nach dem Empfangen im Empfänger ein Wert ausgegeben. In der folgenden Tabelle sind die Differenzen der ermittelten Zeiten eingetragen. Auffällig ist, dass das Routermodul die meiste Zeit noch vor dem Verbindungsaufbau benötigt. Dies könnte mit den Berechnungen in „router_send_data()“ zusammenhängen, die mit in die nachfolgenden Werte eingehen.

Abstand	Router	Core Connector
1	5000	2433
2	5000	
3	5100	
4	5200	
5	5200	

Abbildung 4.7: Übertragungszeit in ns (Komplett)

5 Schlussfolgerung

Die genannten Anforderungen an das neue Routermodul wurden erfüllt. Möchte man allen SpartanMC-Kernen den Datenaustausch ermöglichen, benötigt man beim Core Connector ein vollvermaschtes Netz, beim Router reicht eine Ringtopologie. Bereits ab dem vierten Kern, benötigt die Routertopologie insgesamt im Vergleich zum Duplex Core Connector sowohl weniger Register, als auch weniger LUT. Je mehr Kerne vorhanden sind, umso größer wird der Unterschied. Durch die variablen Wartezeiten der Sender im Falle eines fehlgeschlagenen Verbindungsaufbaus sind Deadlocks praktisch auszuschließen, auch wenn ein vollständiger Beweis hierfür aus Zeitgründen nicht durchgeführt wurde. Die Taktrate kann etwas höher liegen, als beim Duplex Core Connector. Das Javatool „make routing“ nimmt dem Benutzer die Arbeit ab eine Routingtabelle von Hand zu erstellen, was viel Zeit in Anspruch nehmen würde.
Da sowohl die Nachricht, als auch das Headerpaket in den verwendeten Puffern Platz finden müssen, ist die maximal übertragbare Nachrichtenlänge jedoch stark begrenzt. Des Weiteren dauert eine Übertragung länger, da die Verbindung zwischen Sender und Empfänger zuerst aufgebaut werden muss und zwei Buffer erst komplett befüllt werden müssen, bis fortgefahren werden kann. Auch die Berechnung des Headerpakets nimmt Zeit in Anspruch, die der Core Connector nicht benötigt. Bei zwei direkt verbundenen SpartanMC benötigt die Routerlösung zum Verschicken von zehn Werten etwa doppelt so lange.

6 Ausblick

Das Routermodul kann in Zukunft bei zahlreichen Projekten eingesetzt werden, nicht zuletzt durch die umfangreiche Dokumentation in Benutzerhandbuch und Code. Es eignet sich vor allem für kleinere Nachrichten, die nicht besonders schnell übertragen werden müssen, wie zum Beispiel Statusinformationen. Auch eine Weiterentwicklung ist möglich. Einige mögliche Verbesserungen wären:

- Die Wartezeit im Sendermodul sollte zufällig bestimmt werden. Die Routernummer kann als Initialwert dienen.
- Es sollte kontrolliert werden, ob die Verwendung des Arbitermoduls korrekt ist. Möglicherweise wird durch den Reset nach jeder Übertragung die Gleichberechtigung der Eingänge nicht gewährleistet.
- Eine Option ist auch, die Länge der Nachrichten nicht zu beschränken. Hierfür könnte man auf die Buffer verzichten. Nach einem erfolgreichen Verbindungsaufbau würden die Daten dann direkt vom Speicher des Senders zum Speicher des Empfängers übertragen. Da das Warten auf die komplette Nachricht im Buffer in diesem Szenario wegfällt, wäre die Dauer einer Übertragung vermutlich drastisch reduziert.
- Um derzeit einen Router mit mehreren Ausgängen mit einem Router mit mehreren Eingängen zu verbinden, werden in JConfig sogenannte Nets benötigt. JConfig könnte so optimiert werden, dass dieser Umweg nicht mehr notwendig ist.
- Es könnte bewiesen werden, dass in beliebigen Systemen aus mehreren SpartanMC mit Routermodulen keine Deadlocks aufreten können.

7 Anhang

7.1 Abkürzungsverzeichnis

FPGA Field Programmable Gate Array

SpartanMC Spartan Microcontroller

svg Scalable Vector Graphics

LUT Lookup table

LOG2 Zweierlogarithmus

7.2 Abbildungsverzeichnis

7.3 Literaturverzeichnis

[flaig] Bachelorthesis „Design and Implementation of an On-Chip Routing Method for SpartanMC“ von Maximilian Flaig

[man] SpartanMC Usermanual

[tgdi] Folien der Veranstaltung „Technische Grundlagen der Informatik“ von Prof. Dr.-Ing. Andreas Koch aus dem Wintersemester 2011/2012